COMSC-210 Lecture Topic 8
Big Oh and Algorithm Efficiency

Reference
Childs Ch. 9

Counting Numbers Of Operations
operations, comparisons, and assignments *take time*
nested for-loop example:

```
for (int i = 0; i < n; i++)
  for (int j = i + 1; j < n; j++)
    if (a[j] < a[i])
      swap(a[i], a[j]);
```

approximately $n^2/2$ cycles
up to $n^2/2$ swaps

Scaling Data Structure Size
the effect on efficiency: #of operations, etc.
in "nested for-loop" example:
doubling `n` quadruples the time

Online References
Coding Horror: programming and human factors
University Of British Columbia

Performing Simple Timing Studies
measuring elapsed time in "clock ticks":

```
#include <ctime>
...
assert(that the data structure size is n);
clock_t startTime = clock();
do something here...
clock_t endTime = clock();
long elapsedTimeTicks = (long)(endTime - startTime);
```

...may not work for some versions of Linux!

Big Oh -- Predicting Timing Complexity
a way to express how an algorithm scales
in "search loop" example: "O(n)"
to get big-oh:
  write formula for average case #of operations
  simplify the formula,
    so that for entries of `n`
    higher than some number,
    the simplified formula is
    >= the original formula
    times some constant
the answer is not unique!
  so keep it tight

in the search loop:
#of operations = 2 * n
for n >= 1, O(2n) works
  so does O(3n)
for n >= 2, O(n) works
  so does O(2n)
but the tightest is **O(n)**

Operations That Are `O(1)`
measuring elapsed time: independent of "n"
retrieval from an array
adding to end of an array-based list
adding to front of a linked list
retrieving the front of a linked list
retrieving the end of a linked queue
retrieving the 3rd node's entry in a linked list

Operations That Are `O(log n)`
note that O(log n), O(lg n),and O(ln n) are the same: different by a constant multiplier
binary search or bisection

Operations That Are `O(n)`
adding to middle of an array-based list
traversing a list
retrieving a key's value from a linked list
radix sort (a.k.a. bogo sort)

Confirming Big Oh Determinations
measuring elapsed time
try for various entries of `n`

sample test results for $O(n^2)$ operation:

n	n^2	ticks	expected
1000	10^6	1324	1324 (actual)
2000	4×10^6	5436	5296 (1324 x 4)
4000	1.6×10^7	20784	21184 (1324 x 16)
8000	6.4×10^7	86120	84736 (1324 x 64)

"expected" is the row 1 ticks divided by row 1's 2nd column value, times the rows 2nd column valu

Timing Fast Operations
First, increase n to as large as it can be
  e.g., 250 million for the 1st row
  e.g., 2 billion for the 4th row (max int value)
If still too fast, use a loop to repeat the process
  accumulating elapsed time
e.g., using repetitions (thousands; possibly millions!)

```
assert(that the data structure size is n);
clock_t startTime = clock();
for (reps = 0; reps < REPS; reps++) // use any value for REPS
{
  do something here...
}
clock_t endTime = clock();
```

when testing *mutator functions* (setters) that increment or decrement n
...be sure to keep REPS well below the 1st cycle's n so as to not affect n very much

Use Assersions!
be sure to include the `cassert` library
and before each timed operation,
  assert that the data structure size
  is exactly `n`

Using Differential Calculus For Big Oh
df = 0 -> O(1)
df = dn -> O(n)
df = n dn -> O(n-squared)
df = dn/n -> O(log n)
df = log(n) dn -> O(n log n)

<u>O(1) diff. eq.</u>

```
delta-f = 0 // no extra operations...
delta-n = 1 // when the #of elements is increased by 1

delta-f/delta-n = 0/1
df/dn = 0
f(n) = constant
```

<u>O(log n) diff. eq.</u>

```
delta-f = 1 // one more cycle added...
delta-n = n // when the #of elements is doubled

delta-f/delta-n = 1/n
df/dn = 1/n
f(n) = integral of dn/n, or log(n)
```

<u>O(n) diff. eq.</u>

```
delta-f = 1 // one more operaation is added...
delta-n = 1 // when one more element is added

delta-f/delta-n = 1/1
df/dn = 1
f(n) = integral of dn, or n
```

<u>O(n log n) diff. eq.</u>
adding another set of calcs *doubles* the #of elements
so $n = 2^{\#sets}$, log(n) = constant x #sets of calcs
each set of calcs involves n operations

```
delta-f = n x log(n)
```

Operations That Are `O(n log n)`
mergesort, quicksort, heapsort

Operations That Are `O(n²)`
insertion sort, selection sort

"Best Case" Operations
controlling how an operation is done in order to
minimize or reduce its Big Oh behavior.
E.g., insert at END of an array to avoid shifting.
...goes from O(n) to O(1) -- use when creating
a priority queue with large #of entries for testing.
check nested for-loop sort with single loop
before starting sort loops
"average case": no control of how an operation is done.
E.g., insert into array at randomly-chosen key.

```
delta-n = n // when the #of elements is doubled

delta-f/delta-n = n log(n) / n
df/dn = log(n)
f(n) = integral of log(n), or n log(n) - n + constant
```

$O(n^2)$ diff. eq.

```
delta-f = n // add one more full cycle
delta-n = 1 // when onemore element is added

delta-f/delta-n = n/1
df/dn = n
f(n) = integral of n x dn, or n²/2
```

How to perform timing studies -- 4-cycle timing code

```
#include <iostream> // for cout and endl
#include <string> // for string
using namespace std;

#include <cassert> // for assert
#include <cmath> // for log and pow
#include <ctime> // for clock() and clock_t

int main()
{
  // problem setup goes here

  // programmer customizations go here
  int n = 500; // THE STARTING PROBLEM SIZE (MAX 250 MILLION)
  string bigOh = "O(n)"; // YOUR PREDICTION: O(1), O(log n), O(n), O(n log n), or O(n squared)
  const int REPS = 1; // for timing fast operations, use REPS up to 100th of the starting n

  int elapsedTimeTicksNorm = 0;
  double expectedTimeTicks = 0;
  for (int cycle = 0; cycle < 4; cycle++, n*= 2)
  {
    // more problem setup goes here -- the stuff not timed

    // assert that n is the size of the data structure if applicable
    //assert(a.size() == n); // or something like that...

    // start the timer, do something, and stop the timer
    clock_t startTime = clock();
    // do something where n is the "size" of the problem
    clock_t endTime = clock();

    // validation block -- assure that process worked if applicable

    // compute timing results
    long elapsedTimeTicks = (long)(endTime - startTime);
    double factor = pow(2.0, cycle);
    if (cycle == 0)
      elapsedTimeTicksNorm = elapsedTimeTicks;
    else if (bigOh == "O(1)")
      expectedTimeTicks = elapsedTimeTicksNorm;
    else if (bigOh == "O(log n)")
      expectedTimeTicks = log(double(n)) / log(n / factor) * elapsedTimeTicksNorm;
    else if (bigOh == "O(n)")
      expectedTimeTicks = factor * elapsedTimeTicksNorm;
    else if (bigOh == "O(n log n)")
      expectedTimeTicks = factor * log(double(n)) / log(n / factor) * elapsedTimeTicksNorm;
    else if (bigOh == "O(n squared)")
      expectedTimeTicks = factor * factor * elapsedTimeTicksNorm;

    // reporting block
    cout << elapsedTimeTicks;;
    if (cycle == 0) cout << " (expected " << bigOh << ')';
    else cout << " (expected " << expectedTimeTicks << ')';
    cout << " for n=" << n << endl;
} }
```

Example output (for reading n lines from an input text file):

```
1436 (expected O(n)) for n=500
2742 (expected 2872) for n=1000
5442 (expected 5744) for n=2000
10828 (expected 11488) for n=4000
```